



下载APP



10 | Flow，是异步编程的终极选择吗？

2021-12-06 范学雷

《深入剖析Java新特性》

课程介绍 >



讲述：范学雷

时长 19:20 大小 17.71M



你好，我是范学雷。今天，我们讨论反应式编程。

反应式编程曾经是一个很热门的话题。它是代码的控制的一种模式。如果不分析其他的模式，我们很难识别反应式编程的好与坏，以及最合适它的使用场景。所以，我们今天的讨论，和以往有很大的不同。

除了反应式编程之外，我们还会花很大的篇幅讨论其他的编程模式，包括现在的和未来的。希望这样的安排，能够帮助你根据具体的场景，选择最合适的模式。



我们从阅读案例开始，先来看一看最传统的模式，然后一步一步地过渡到反应式编程，最后我们再来稍微聊几句 Java 尚未发布的协程模式。

阅读案例

我想，你和我一样，无论是学习 C 语言，还是 Java 语言，都是从打印 "Hello, world!" 这个简单的例子开始的。我们再来看看这个我们熟悉的代码。

```
1 System.out.println("Hello, World!");
```

[复制代码](#)

这段代码就是使用了最常用的代码控制模式：指令式编程模型。**所谓指令式编程模型，需要我们通过代码发布指令，然后等待指令的执行以及指令执行带来的状态变化。我们还要根据目前的状态，来确定下一次要发布的指令，并且用代码把下一个指令表示出来。**

上面的代码里，我们发布的指令就是：标准输出打印 "Hello, World!" 这句话。然后，我们就等待指令的执行结果，验证我们编写的代码有没有按照我们的指令工作。

指令式编程模型

指令式编程模型关注的重点就在于控制状态。"Hello, world!" 这个例子能看出来一点端倪，但是要了解状态变化和控制，我们需要看两行以上的代码。

```
1 try {
2     Digest messageDigest = Digest.of("SHA-256");
3     byte[] digestValue =
4         messageDigest.digest("Hello, world!".getBytes());
5 } catch (NoSuchAlgorithmException ex) {
6     System.out.println("Unsupported algorithm: SHA-256");
7 }
```

[复制代码](#)

在上面的这段代码里，我们首先调用 `Digest.of` 方法，得到一个 `Digest` 实例；然后调用这个实例的方法 `Digest.digest`，获得一个返回值。第一个方法执行完成后，获得了第一个方法执行后的状态，第二个方法才能接着执行。

这种顺序执行的模式，逻辑简单直接。简单直接本身就有着巨大的能量，特别是实现精确控制方面。所以，这种模式在通用编程语言设计和一般的应用程序开发中，占据着压倒性的优势。

但是，这种模式需要维护和同步状态。如果状态数量大，我们就要把大的代码块分解成小的代码块；这样，我们编写的代码才能更容易阅读，更容易维护。而更大的问题来自于状态同步需要的顺序执行。

比如说吧，上面的例子中，Digest.of 这个方法实现，可能效率很高，执行得很快；而 Digest.digest 这个方法的实现，它的执行速度可能就是毫秒级的，甚至是秒一级别的。在要求低延迟、高并发的环境下，等待 Digest.digest 调用的返回结果，可能就不是一个好的选择。换句话说，阻塞在方法的调用上，增加了系统的延迟，降低了系统能够支持的吞吐量。

这种顺序执行的模式带来的延迟后果，在互联网时代的很多场景下是无法忍受的（比如春节的火车票预售系统，或者网上购物节的订购系统等）。存在这种问题最典型的场景之一，就是客户端 - 服务器这种架构下的传统的套接字编程接口。它也引发了大约 20 年前提出的 C10K 问题（支持 1 万个并发用户）。

怎样解决 C10K 问题呢？一个主要方向，就是使用非阻塞的异步编程。

声明式编程模型

非阻塞的异步编程，并不是可以通过编程语言或者标准类库就可以得到的。支持非阻塞的异步编程，需要大幅度地更改代码，转换代码编写的思维习惯。

我们可以使用打电话来做个比方。

传统的指令式编程模型，就像我们通常打电话一样。我们拨打对方的电话号码，然后等待接听，然后通话，然后挂断。当我们挂断电话的时候，打电话这一个过程也就结束了，我们也拿到了想要的结果。

而非阻塞的异步编程，更像是电话留言。我们拨打对方的电话，告诉对方方便的时候，回拨电话，然后就挂断了。当我们挂断电话的时候，打电话这一个过程当然也是结束了，但是我们没有拿到想要的结果。想要的结果，还要依靠回拨电话，才能够得到。

而类似于回拨电话的逻辑，正是非阻塞的异步编程的关键模型。映射到代码上，就是使用回调函数或者方法。

当我们试图使用回调函数时，我们编写代码的思想和模型都会产生巨大的变化。我们关注的重点，就会从指令式编程模型的“控制状态”转变到“控制目标”。这时候，我们编程模型也就转变到了**声明式的编程模型**。

如果指令式编程模型的逻辑是告诉计算机“该怎么做”，那么声明式的编程模型的逻辑就是告诉计算机“要做什么”。指令式编程模型的代码像是流水线作业的工程师，事无巨细，拧好每一个螺丝；而声明式的编程模型的代码，更像是稳坐在军帐中的军师，布置任务，运筹帷幄。

我们前面讨论的 Digest，能不能实现非阻塞的异步编程呢？答案是肯定的，不过我们需要彻底地更改代码，从 API 到实现都要转换思路。下面这段代码里声明的 API，就是我们尝试使用声明式编程的一个例子。

[复制代码](#)

```
1 public sealed abstract class Digest {
2     public static void of(String algorithm,
3         Consumer<Digest> onSuccess, Consumer<Integer> onFailure) {
4         // snipped
5     }
6
7     public abstract void digest(byte[] message,
8         Consumer<byte[]> onSuccess, Consumer<Integer> onFailure);
9 }
```

转化了思路的 Digest.of 方法，就像是布置任务：如果执行成功，请继续执行 A 计划（也就是 onSuccess 这个回调函数）；否则，就继续执行 B 计划（也就是 onFailure 这个回调函数）。其实，这也就是我们前面提到的，告诉计算机“要做什么”的概念。

有了回调函数的设计，代码的实现方式就放开了管制。无论是回调函数的实现，还是回调函数的调用，都可以自由地选择是采用异步的模式，还是同步的模式。不用说，这种自由很具有吸引力。从 JDK 7 引入 NIO 新特性开始，这种模式开始进入 Java 的工业实践，并且取得了巨大的成功。出现了一大批的明星项目。

不过，回调函数的设计也有着天生的缺陷。这个缺陷，就是回调地狱（Callback Hell，常被译为回调地狱。为了更直观地表达，我更喜欢把它叫做回调堆挤）。什么意思呢？通常

地，我们需要布置多个小的任务，才能完成一项大的任务。这些小任务还有可能是有因果关系的任务，这时候，就需要小任务的配合，或者按顺序执行。

比如说，上面的 Digest 设计，我们先要判断 of 方法能不能成功；如果成功的话，那么就使用这个 Digest 实例，调用它的 Digest.digest 方法。而 Digest.digest 方法的调用，也要作出 A 计划和 B 计划。这样，两个回调函数的使用，就会堆积起来。如果回调函数的嵌套增多，代码看起来就像挤在一块一样，形式上不美观，阅读起来很费解，维护起来难度很大。

下面的这段代码，就是我们使用回调函数设计的 Digest 的一个用例。这个用例里，回调函数的嵌套仅仅有两层，代码的形式已经变得很难阅读了。你可以尝试编写一个 3 层或者 5 层的回调函数的嵌套，体验一下深度嵌套的代码是什么样子的。

[复制代码](#)

```
1 Digest.of("SHA-256",
2     md -> {
3         System.out.println("SHA-256 is not supported");
4         md.digest("Hello, world!".getBytes(),
5             values -> {
6                 System.out.println("SHA-256 is available");
7             },
8             errorCode -> {
9                 System.out.println("SHA-256 is not available");
10            });
11     },
12     errorCode -> {
13         System.out.println("Unsupported algorithm: SHA-256");
14     });
```

如果说，回调函数带来的形式的堆积我们还可以克服的话；那这种形式上的堆积带来的逻辑堆积，我们就几乎不可承受了。**逻辑上的堆积，意味着代码的深度耦合。而深度耦合，意味着代码维护困难。深度嵌套里的一点点代码修改，都可能通过嵌套层层朝上传递，最后牵动全局。**

这就导致，使用回调函数的声明式编程模型有着严重的场景适应问题。我们通常只使用回调函数解决性能影响最大的模块，比如说网络数据的传输；而大部分的代码，依然使用传统的、顺序执行的指令式模型。

好在，业界也有很多努力，试图改善回调函数的使用困境。其中最出色也是影响最大的一个，就是反应式编程。

反应式编程

反应式编程的基本逻辑，仍然是告诉计算机“要做什么”；但是它的关注点转移到了数据的变化以及数据和变化的传递上，或者说，是转移到了对数据变化的反应上。所以，**反应式编程的核心是数据流和变化传递**。

如果我们从数据的流向角度来看的话，数据有两种基本的形式：数据的输入和数据的输出。从这两种基本的形式，能够衍生出三种过程：最初的来源，数据的传递和最终的结局。

数据的输出

在 Java 的反应式编程模型的设计里，数据的输出使用只有一个参数的 `Flow.Publisher` 来表示。

 复制代码

```
1 @FunctionalInterface
2 public static interface Publisher<T> {
3     public void subscribe(Subscriber<? super T> subscriber);
4 }
```

在 `Flow.Publisher` 的接口设计里，泛型 `T` 表示的就是数据的类型。数据输出的对象，是使用 `Flow.Subscriber` 来表示的。换句话说，数据的发布者通过授权订阅者，来实现数据从发布者到订阅者的传递。一个数据的发布者，可以有多个数据的订阅者。

需要注意的是，订阅的接口，安排在了 `Flow.Publisher` 这个接口里。这也就意味着，订阅者的订阅行为，是由数据的发布者发起的，而不是订阅者发起的。

数据最初的来源，就是一种形式的数据输出；它只有数据输出这一个传递方向，而不能接收数据的输入。

比如下面的代码，就是一个表示数据最初来源的例子。在这段代码里，数据的类型是字节数组；而数据发布的实现，我们使用了 Java 标准类库的参考性实现 `SubmissionPublisher`

这个类。

 复制代码

```
1 SubmissionPublisher<byte[]> publisher = new SubmissionPublisher<>();
```

数据的输入

下面，我们再来看下数据的输入。

在 Java 的反应式编程模型的设计里，数据的输入用只有一个参数的 `Flow.Subscriber` 来表示。也就是我们前面提到的订阅者。

 复制代码

```
1 public static interface Subscriber<T> {  
2     public void onSubscribe(Subscription subscription);  
3  
4     public void onNext(T item);  
5  
6     public void onError(Throwable throwable);  
7  
8     public void onComplete();  
9 }
```

在 `Flow.Subscriber` 的接口设计里，泛型 `T` 表示的就是数据的类型。这个接口里一共定义了四种任务，并分别规定了下面四种情形下的反应：

1. 如果接收到订阅邀请该怎么办？这个行为由 `onSubscribe` 这个方法的实现确定。
2. 如果接收到数据该怎么办？这个行为由 `onNext` 这个方法的实现确定。
3. 如果遇到了错误该怎么办？这个行为由 `onError` 这个方法的实现确定。
4. 如果数据传输完毕该怎么办？这个行为由 `onComplete` 这个方法的实现确定。

数据最终的结局，就是一种形式的数据输入；它只有数据输入这一个传递方向，而不能产生数据的输出。

比如下面的代码，就是一个表示数据最终结果的例子。在这段代码里，我们使用一个泛型来表示数据的类型；然后，使用了一个 Consumer 函数来表示我们该怎么处理接收到的数据。这样的安排让这个例子具有了普遍的意义。只要稍作修改，就可以把它使用到实际场景中去了。

 复制代码

```
1 package co.ivijus.flow.reactive;
2
3 import java.util.concurrent.Flow;
4 import java.util.function.Consumer;
5
6 public class Destination<T> implements Flow.Subscriber<T>{
7     private Flow.Subscription subscription;
8     private final Consumer<T> consumer;
9
10    public Destination(Consumer<T> consumer) {
11        this.consumer = consumer;
12    }
13
14    @Override
15    public void onSubscribe(Flow.Subscription subscription) {
16        this.subscription = subscription;
17        subscription.request(1);
18    }
19
20    @Override
21    public void onNext(T item) {
22        subscription.request(1);
23        consumer.accept(item);
24    }
25
26    @Override
27    public void onError(Throwable throwable) {
28        throwable.printStackTrace();
29    }
30
31    @Override
32    public void onComplete() {
33        System.out.println("Done");
34    }
35 }
```

数据的控制

你可能已经注意到了，Flow.Subscriber 接口，并没有和 Flow.Publisher 直接联系。取而代之地出现了一个中间代理 Flow.Subscription。Flow.Subscription 管理、控制着

Flow.Publisher 和 Flow.Subscriber 之间的连接，以及数据的传递。

也就是说，在 Java 的反应式编程模型里，数据的传递控制从数据和数据的变化里分离了出来。这样的分离，对于降低功能之间的耦合意义重大。

[复制代码](#)

```
1 public static interface Subscription {  
2     public void request(long n);  
3  
4     public void cancel();  
5 }
```

在 Flow.Subscription 的接口设计里，我们定义了两个方法。一个方法表示订阅者希望接收的数据数量，也就是 Subscription.request 这个方法。另一个方法表示订阅者希望取消订阅，也就是 Subscription.cancel 这个方法。

数据的传递

除了最初的来源和最终的结局，数据表现还有一个过程，就是数据的传递。数据的传递这个过程，既包括接收输入数据，也包括发送输出数据。在数据传递这个环节，数据的内容可能会发生变化，数据的数量也可能会发生变化（比如，过滤掉一部分的数据，或者修改输入的数据，甚至替换掉输入的数据）。

在 Java 的反应式编程模型的设计里，这样的过程是由 Flow.Processor 表示的。Flow.Processor 是一个扩展了 Flow.Publisher 和 Flow.Subscriber 的接口。所以，Flow.Processor 有两个数据类型，泛型 T 表述输入数据的类型，泛型 R 表述输出数据的类型。

[复制代码](#)

```
1 public static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {  
2 }
```

下面的代码，就是一个表示数据传递的例子。在这段代码里，我们使用泛型来表示输入数据和输出数据的类型；然后，我们使用了一个 Function 函数，来表示该怎么处理接收到的

数据，并且输出处理的结果。这样的安排让这个例子具有了普遍的意义。稍作修改，你就可以把它用到实际场景中去了。

 复制代码

```
1 package co.ivijus.flow.reactive;
2
3 import java.util.concurrent.Flow;
4 import java.util.concurrent.SubmissionPublisher;
5 import java.util.function.Function;
6
7 public class Transform<T, R> extends SubmissionPublisher<R>
8     implements Flow.Processor<T, R> {
9     private Function<T, R> transform;
10    private Flow.Subscription subscription;
11
12    public Transform(Function<T, R> transform) {
13        super();
14        this.transform = transform;
15    }
16
17    @Override
18    public void onSubscribe(Flow.Subscription subscription) {
19        this.subscription = subscription;
20        subscription.request(1);
21    }
22
23    @Override
24    public void onNext(T item) {
25        submit(transform.apply(item));
26        subscription.request(1);
27    }
28
29    @Override
30    public void onError(Throwable throwable) {
31        closeExceptionally(throwable);
32    }
33
34    @Override
35    public void onComplete() {
36        close();
37    }
38 }
```

过程的串联

既然数据的表述方式分为输入和输出两种基本的形式，而且还提供了由此衍生出来的三种过程，我们就能够把数据的处理过程，很方便地串联起来了。

下面的代码，就是我们试图把最初的来源、数据的传递和最终的结局这三个过程，串联成一个更大的过程的例子。当然，你也可以试着串联进更多的数据处理过程。

 复制代码

```
1 private static void transform(byte[] message,
2     Function<byte[], byte[]> transformFunction) {
3     SubmissionPublisher<byte[]> publisher =
4         new SubmissionPublisher<>();
5
6     // Create the transform processor
7     Transform<byte[], byte[]> messageDigest =
8         new Transform<>(transformFunction);
9
10    // Create subscriber for the processor
11    Destination<byte[]> subscriber = new Destination<>((
12        values -> System.out.println(
13            "Got it: " + Utilities.toHexString(values)));
14
15    // Chain processor and subscriber
16    publisher.subscribe(messageDigest);
17    messageDigest.subscribe(subscriber);
18    publisher.submit(message);
19
20    // Close the submission publisher.
21    publisher.close();
22 }
```

串联的形式，接藕了不同环节的关联；而且每个环节的代码也可以换个场景复用。支持过程的串联，是反应式编程模型强大的最大动力之一。像 Scala 这样的编程语言，甚至把过程串联提升到了编程语言的层面来支持。这样做，毫无疑问大幅度地提高了编码的效率和代码的美观程度。

简洁的重构

介绍完 Java 的反应式编程模型设计，我们要回头看看我们在阅读案例里提出的问题了。反应式编程，是怎么解决顺序执行的模式带来的延迟后果的呢？反应式编程，怎么解决回调函数带来的堆挤问题呢？

我们还是先看一眼使用反应式编程模型的代码，然后再来讨论这些问题吧。下面的代码，就是我们对阅读案例里 Digest 用法的改进。

```
1 Returned<Digest> rt = Digest.of("SHA-256");
2 switch (rt) {
3     case Returned.ReturnValue rv -> {
4         // Get the returned value
5         if (rv.returnValue() instanceof Digest d) {
6             // Call the transform method for the message digest.
7             transform("Hello, World!".getBytes(), d::digest);
8
9             // Wait for completion
10            Thread.sleep(20000);
11        } else { // unlikely
12            System.out.println("Implementation error: SHA-256");
13        }
14    }
15    case Returned.ErrorCode ec ->
16        System.out.println("Unsupported algorithm: SHA-256");
17 }
18
```

在这个例子里，我们没有发现类似于回调函数一样的堆挤现象。这里面，起重要作用的就是我们上面提到的过程的串联这种形式。Java 的反应式编程模型里的过程串联和数据控制的设计，以及数据输入和输出的分离，降低了代码的耦合，不再需要嵌套的调用了。

在这个例子里，我们还看到了 Digest.digest 方法的直接使用。为了能够使用反应式编程模型，我们没有必要去修改 Digest 代码。只要把 Digest 原来的设计和实现，恰当地放到反应式编程模型里来，就能够实现异步非阻塞的设想了。这一点，无疑具有极大的吸引力。如果不是被逼无奈，谁会去颠覆已有的代码呢？

那到底反应式编程模型是怎么支持异步非阻塞的呢？其实，和回调函数一样，反应式编程既能够支持同步阻塞的模式，也能够支持异步非阻塞的模式。如果这些接口实现是异步非阻塞模式的，这些实现的调用，也就是异步非阻塞的。当然，反应式编程模型的主要使用场景，目前还是异步非阻塞模式。

比如我们例子中的 SubmissionPublisher，就是一个异步非阻塞模式的实现。在上面的代码里，如果没有调用 Thread.sleep，我们可能还看不到 Digest 的处理结果，主线程就退出了。这就是一个非阻塞的实现表现出来的现象。

缺陷与对策

到目前为止，反应式编程模型看起来还很完美。可是，反应式编程模型的缺陷也很要命。其中最要命的缺陷，就是错误很难排查，这是异步编程的通病。而反应式编程模型的解耦设计，加剧了错误排查的难度，这会严重影响开发的效率，降低代码的可维护性。

目前来看，解决反应式编程模型的缺陷，或者说是异步编程的缺陷的方向，似乎又要回到了指令式编程模型这条老路上来了。这里最值得提及的就是协程（Fiber）这个概念（目前，Java 的协程模式还没有发布，但是我可以带你先了解一下）。

我们再来看看阅读案例里提到的这段代码。为了方便你阅读，我把它拷贝粘贴到这里来了。

[复制代码](#)

```
1 try {
2     Digest messageDigest = Digest.of("SHA-256");
3     byte[] digestValue =
4         messageDigest.digest("Hello, world!".getBytes());
5 } catch (NoSuchAlgorithmException ex) {
6     System.out.println("Unsupported algorithm: SHA-256");
7 }
```

在 Java 的指令式编程模型里，这段代码要在一个线程里执行。我们首先调用 `Digest.of` 方法，得到一个 `Digest` 实例；然后调用这个实例的方法 `Digest.digest`，获得一个返回值。在每个方法返回之前，线程都会处于等待状态。而线程的等待，是造成资源浪费的最大因素。

而协程的处理方式，消除了线程的等待。如果调用阻塞，就会把资源切换出去，执行其他的操作。这就节省了大量的计算资源，使得系统在阻塞的模式下，支持大规模的并发。如果指令式编程模型能够通过协程的方式支持大规模的并发，也许它是一个颠覆现有高并发架构的新技术。

目前，Java 的协程模式还没有发布。它能够给反应式编程模型带来什么样的影响，能够给我们实现大规模并发系统带来多大的便利？这些问题的答案，我们还需要等待一段时间。

总结

好，到这里，我来做个小结。前面，我们讨论了指令式编程模型和声明式编程模型，回调函数以及回调地狱，以及 Java 反应式编程模型的基本组件。

限于篇幅，我们不能展开讨论 Java 反应式编程模型的各种潜力和变化，比如“反应式宣言”“背压”这样的热门词汇。我建议你继续深入地了解反应式编程的这些要求（比如反应式宣言和反应式系统），以及成熟的明星产品（比如 Akka 和 Spring 5+）。

由于 Java 的协程模式还没有发布，我对反应式编程的未来还没有清晰的判断。也欢迎你在留言区里留言、讨论反应式编程的现在和未来。

另外，我还拎出了几个今天讨论过的技术要点，这些都可能在你面试中出现哦。通过这一次学习，你应该能够：

了解指令式编程模型和声明式编程模型这两个术语；

- 面试问题：你知道声明式编程模型吗，它是怎么工作的？

了解 Java 反应式编程模型的基本组件，以及它们的组合方式；

- 面试问题：你怎么使用 Java 反应式编程模型？

知道回调函数的形式，以及回调地狱这个说法。

- 面试问题：你知道回调函数有什么问题吗？

反应式编程是目前主流的支持高并发的技术架构思路。学会反应式编程，意味着你有能力处理高并发应用这样的需求。能够编写高并发的代码，现在很重要，以后更重要。学会使用 Java 反应式编程模型这样一个高度抽象的接口，毫无疑问能够提升你的技术深度。

思考题

今天的思考题，我们来试着使用一下 Java 反应式编程模型。在讨论反应式编程的时候，计算 $a = b + c$ 是一个常用的范例。在这个计算里， b 和 c 随着时间的推移，会发生变化。而每一次的变化，都会影响 a 的计算结果。

现在我们假设 a 表示的数据是一件事情结束的时候是星期几， b 表示的数据是一件事情开始的时候是星期几， c 表示处理完这件事情需要多少天。你会怎么使用 Java 反应式编程模型来处理这个问题？

欢迎你在留言区留言、讨论，分享你的阅读体验以及你的设计和代码。我们下节课见！

注：本文使用的完整的代码可以从 [🔗 GitHub](#) 下载，你可以通过修改 [🔗 GitHub](#) 上 [🔗 review template](#) 代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在 [🔗 反应式编程专用的代码评审目录](#) 下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 flow/review/xuelei 的目录下面。

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

 生成海报并分享

 赞 1

 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | 异常恢复，付出的代价能不能少一点？

下一篇 11 | 矢量运算：Java 的机器学习要来了吗？

精选留言 (9)

 写留言



Calvin

2021-12-07

老师，这个感觉有点抽象和复杂，有没有什么现成的已经在使用的业务场景的例子可以介绍来看看吗？加深下印象。

PS：思考题 PR：<https://github.com/Xueleifan/java-up/pull/18>

展开 ▾

作者回复：有时间看看 Apache Spark, Akka, 或者 Kafka。现在主流的高并发架构里，都有反应式的影子在。



 1

**aoe**

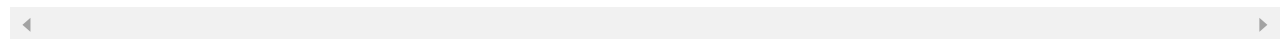
2021-12-07

虽然反应式编程编写起来复杂，但是因其是针对并发场景设计的，有机会还是很想体验一下威力，因为：

1. 数据传递时具有不可变的特性，天然支持线程安全；
2. 令人崩溃的多线程操作由 JDK 团队负责处理，降低了并发编程门槛...

展开 ∨

作者回复: 是的，现在看，反应式编程是高并发场景的主流选择。



共 5 条评论 >

👍 1

**许灵**

2021-12-10

<https://github.com/Xueleifan/java-up/pull/19>

反应式编程里，这个实现可能还有些问题。希望老师指点，这里随着数据变更，如果没有sleep会出现计算结果相同的问题，不知道有没有好的处理方式。

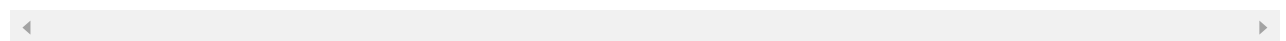
**bigben**

2021-12-09

感觉这个api设计的不好理解也不合理，我觉得subscribe方法应该在subscriber上面，而不是publisher上面，太别扭了！

展开 ∨

作者回复: 其实，这也许恰恰是精妙的地方。想一想数据流和控制权，可能感觉就好一点了。

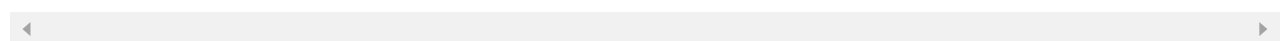
**worry**

2021-12-08

go语言本身就支持协程，go语言在云服务上的成功是不是能证明协程这种模式是未来呢。spring是强烈推荐反应式编程模式的。

展开 ∨

作者回复: 我想是的，但是还需要时间验证。





2021-12-07

协程看起来很强, 不知道啥时候能用上呢

展开 ∨

作者回复: 下一个长期支持版可能就看到苗头了。

**ABC**

2021-12-07

Java 的虚拟线程,尾调用优化等来自

OpenJDK的 loom 项目。刚去看了下,目前依然只支持 Mac 和 Linux 系统。老师,如果正式发布了,是否会内置到 JDK 中呢?

作者回复: 是的, 会是JDK的一部分。

**ABC**

2021-12-07

换成反应式增加了不少代码。之前看过一些Spring提供的反应式编程, 但碍于生态问题, 那会很难用到生产环境中, 现在也许成熟多了。

在JavaScript里面, 很早就用异步请求了, 但很简单和方便。

...

展开 ∨

作者回复: 异步一定要再封装一层, 否则太难用了。JDK没有提供进一步的封装, 就显得比较复杂。

**kimoti**

2021-12-06

感觉原来三行代码,换成响应式复杂了许多

展开 ∨

作者回复: 的确要复杂很多，不过可以再抽象出来一层简化。如果你看看Akka的设计，总体感觉是应用层的代码可以更少、更简单。

